Assignment 5

Anand Uday Gokhale, EE17B158

March 3 2019

1 Abstract

This week we wish to solve for the currents in a resistor. The currents depend on the shape of the resistor. We also want to know which part of the resistor is likely to get hottest.

2 Introduction

A cylindrical wire is soldered to the middle of a copper plate and its voltage is held at 1 Volt. One side of the plate is grounded, while the remaining are floating. The plate is 1 cm by 1 cm in size.
We shall use these equations:
The Continuity Equation:

$$\nabla \cdot \vec{j} = -\frac{\partial \rho}{\partial t} \tag{1}$$

Ohms Law:

$$\vec{j} = \sigma \vec{E} \tag{2}$$

The above equations along with the definition of potential as the negative gradient of Field give:

$$\nabla^2 \phi = \frac{1}{\rho}\frac{\partial \rho}{\partial t} \tag{3}$$

For DC Currents, RHS of equation (3) is 0. Hence:

$$\nabla^2 \phi = 0 \tag{4}$$

3 Assignment 3

3.1 Defining Parameters

We have chosen a 25x25 grid with a circle of radius 8 centrally located maintained at $V = 1V$ by default. We also choose to run the difference equation for 1500 iterations by default

```
if(len(sys.argv)==5):
    Nx=int(sys.argv[1])
    Ny=int(sys.argv[2])
    radius=int(sys.argv[3])
    Niter=int(sys.argv[4])
    print("Using user provided params")
else:
    Nx=25 # size along x
    Ny=25 # size along y
    radius=8 #radius of central lead
    Niter=1500 #number of iterations to perform
    print("Using default Parameters")
```

3.2 Initializing Potential

We start by creating an zero 2-D array of size Nx x Ny. then a list of coordinates lying within the radius is generated and these points are initialized to 1.

```
phi=np.zeros((Nx,Ny),dtype = float)
x,y=np.linspace(-0.5,0.5,num=Nx,dtype=float),np.linspace(-0.5,0.5,num=Ny,dtype=float)
Y,X=np.meshgrid(y,x,sparse=False)
phi[np.where(X**2+Y**2<(0.35)**2)]=1.0
plt.xlabel("X")
plt.ylabel("Y")
plt.contourf(X,Y,phi)
plt.colorbar()
plt.show()
```

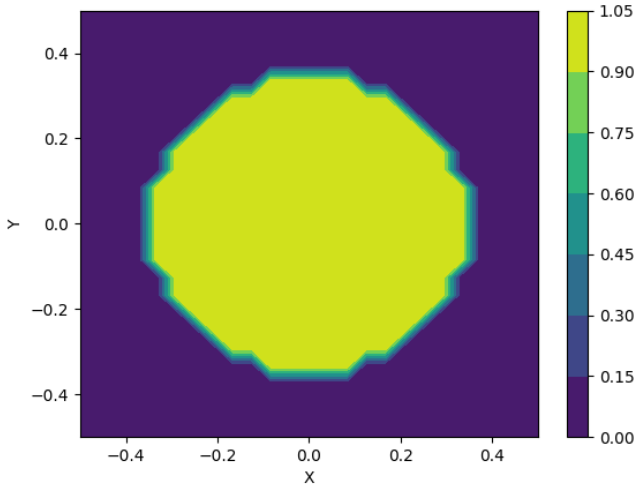

Figure 1: Initial Potential

3.3 Performing Iterations

3.3.1 Updating Potential

We use Equation(4) to do this.But Equation (4) is a differential equation. We need to first convert it to a difference equation as all of our code is in discrete domain. We write it as :

$$\phi_{i,j} = 0.25 * (\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j+1} + \phi_{i,j-1}) \tag{5}$$

```
def update_phi(phi,phiold):
    phi[1:-1,1:-1]=0.25*(phiold[1:-1,0:-2]+ phiold[1:-1,2:]+
                         phiold[0:-2,1:-1]+ phiold[2:,1:-1])
    return phi
```

3.3.2 Applying Boundary Conditions

The bottom boundary is grounded. The other 3 boundaries have a normal potential of 0

```
def boundary(phi,mask = np.where(X**2+Y**2<(0.35)**2)):
    phi[1:-1,0]=phi[1:-1,1] # Left Boundary
    phi[1:-1,Nx-1]=phi[1:-1,Nx-2] # Right Boundary
    phi[0,1:-1]=phi[1,1:-1] # Top Boundary
    phi[Ny-1,1:-1]=0
    phi[mask]=1.0
    return phi
```

3.3.3 Calculating error and running iterations

```
for k in range(Niter):
    phiold = phi.copy()
    phi = update_phi(phi,phiold)
    phi = boundary(phi)
    err[k] = np.max(np.abs(phi-phiold))
```

3.3.4 Plotting the errors

We will plot the errors on semi-log and log-log plots. We note that the error falls really slowly and this is one of the reasons why this method of solving the Laplace equation is discouraged

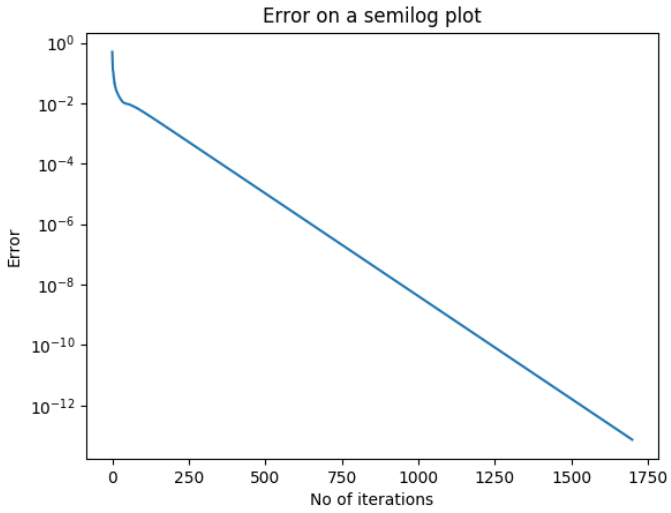

Figure 2: Semi-log plot of error

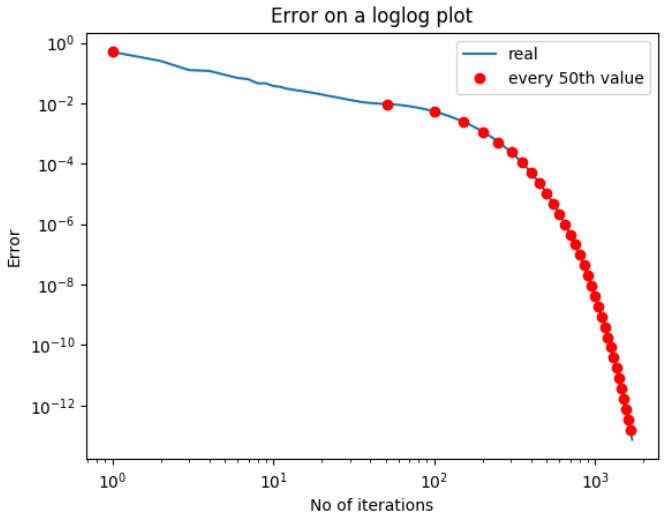

Figure 3: Log-log plot of error

3.4 Fitting the error

We note that the error is decaying exponentially for higher iterations.I have plotted 2 fits. One considering all the iterations(fit1) and another without considering the first 500 iterations. There is very little difference between the two fits

```
def get_fit(y,Niter,lastn=0):
    log_err = np.log(err)[-lastn:]
    X = np.vstack([(np.arange(Niter)+1)[-lastn:],np.ones(log_err.shape)]).T
    log_err = np.reshape(log_err,(1,log_err.shape[0])).T
    return s.lstsq(X, log_err)[0]


def plot_error(err,Niter,a,a_,b,b_):
    plt.title("Best fit for error on a loglog scale")
    plt.xlabel("No of iterations")
    plt.ylabel("Error")
    x = np.asarray(range(Niter))+1
    plt.loglog(x,err)
    plt.loglog(x[::100],np.exp(a+b*np.asarray(range(Niter)))[::100],'ro')
    plt.loglog(x[::100],np.exp(a_+b_*np.asarray(range(Niter)))[::100],'go')
    plt.legend(["errors","fit1","fit2"])
    plt.show()

b,a = get_fit(err,Niter)
b_,a_ = get_fit(err,Niter,500)
plot_error(err,Niter,a,a_,b,b_)
```

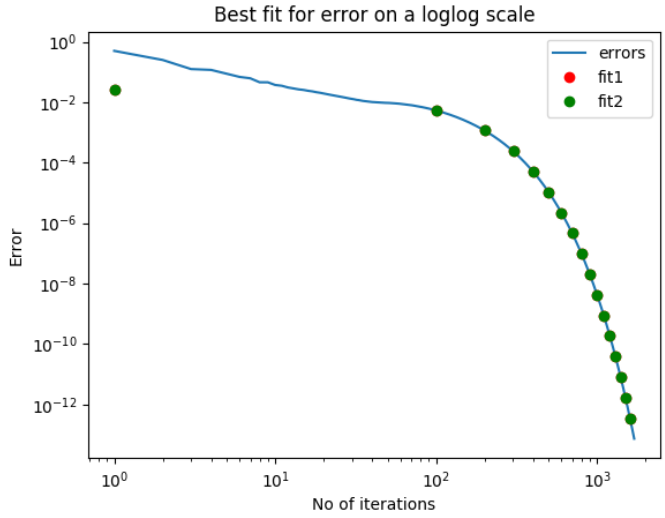

Figure 4: Best Fit of error

3.5 Plotting Maximum Possible Error

This method of solving Laplace's Equation is known to be one of the worst available. This is because of the very slow coefficient with which the error reduces.

```
def find_net_error(a,b,Niter):
    return -a/b*np.exp(b*(Niter+0.5))
b,a = get_fit(err,Niter)
b_,a_ = get_fit(err,Niter,500)
plot_error(err,Niter,a,a_,b,b_)
iter=np.arange(100,1501,100)
plt.grid(True)
plt.title(r'Plot of Cumulative Error values On a loglog scale')
plt.loglog(iter,np.abs(find_net_error(a_,b_,iter)),'ro')
plt.xlabel("iterations")
plt.ylabel("Net  maximum error")
plt.show()
```

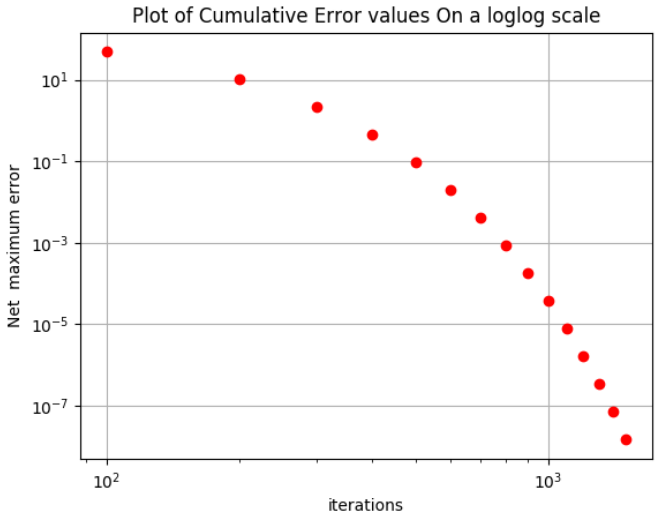

Figure 5: Cumulative error values on a log log scale

3.6 Plotting ϕ

```
fig1=plt.figure(4)       # open a new figure
ax=p3.Axes3D(fig1) # Axes3D is the means to do a surface plot
plt.title('The 3-D surface plot of the potential')
surf = ax.plot_surface(Y, X, phi.T, rstride=1, cstride=1, cmap=plt.cm.jet)
plt.show()

plt.title("2D Contour plot of potential")
```

```
plt.xlabel("X")
plt.ylabel("Y")
plt.contourf(Y,X[::-1],phi)
plt.colorbar()
plt.show()
```

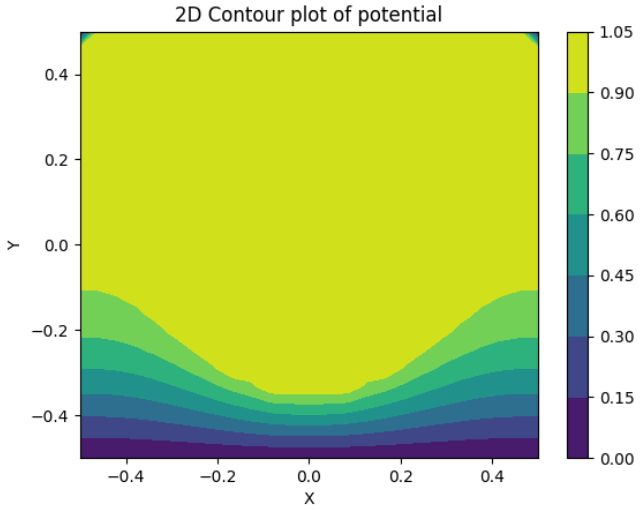

Figure 6: 2d Plot of Potential

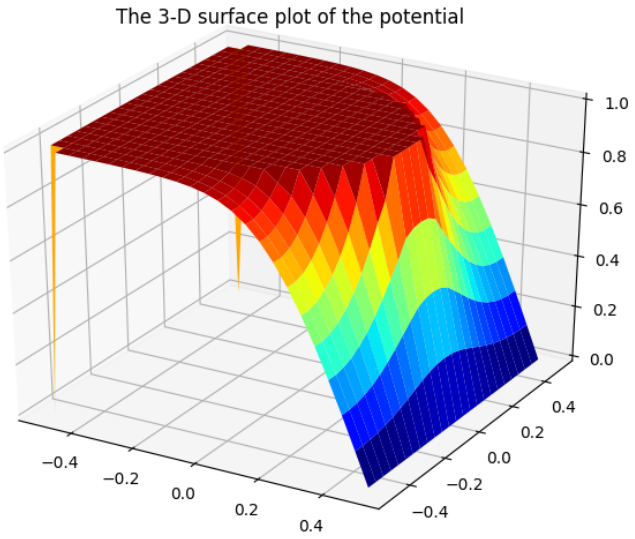

Figure 7: 3d Plot of Potential

3.7 Finding and Plotting J

$$J_{x,ij} = 0.5 * (\phi_{i,j-1} - \phi_{i,j+1}) \tag{6}$$

$$J_{y,ij} = 0.5 * (\phi_{i-1,j} - \phi_{i+1,j}) \tag{7}$$

```
Jx,Jy = (1/2*(phi[1:-1,0:-2]-phi[1:-1,2:]),1/2*(phi[:-2,1:-1]-phi[2:,1:-1]))

plt.title("Vector plot of current flow")
plt.quiver(Y[1:-1,1:-1],-X[1:-1,1:-1],-Jx[:,::-1],-Jy)
x_c,y_c=np.where(X**2+Y**2<0.35**2)
plt.plot((x_c-Nx/2)/Nx,(y_c-Ny/2)/Ny,'ro')
plt.show()
```

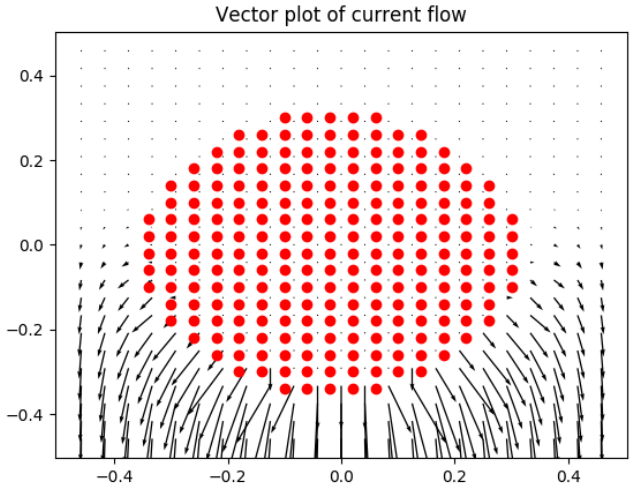

Figure 8: Vector plot of current flow

4 Conclusion

We have used discrete differentiation to solve Laplace's Equations.This method of solving Laplace's Equation is known to be one of the worst available. This is because of the very slow coefficient with which the error reduces.